

JS Level 1



Введение



Условия и функции

В этой лекции мы с вами поговорим о ключевых конструкциях языка, которые позволят нам писать более интересные программы.

Эта лекция будет "потяжелее", чем предыдущие (далее "проще" точно не будет), поэтому привыкайте к синтаксису (правилам написания) и будьте внимательны: JS – это язык программирования, каждый символ здесь имеет своё значение, если вы неправильно напишете хотя бы один символ, то приложение работать не будет, причём узнаете вы об этом, к сожалению, только после того, как начнёте тестировать приложение (если, конечно, не умеете пользоваться специальными инструментами, но об этом позже).



Повторение



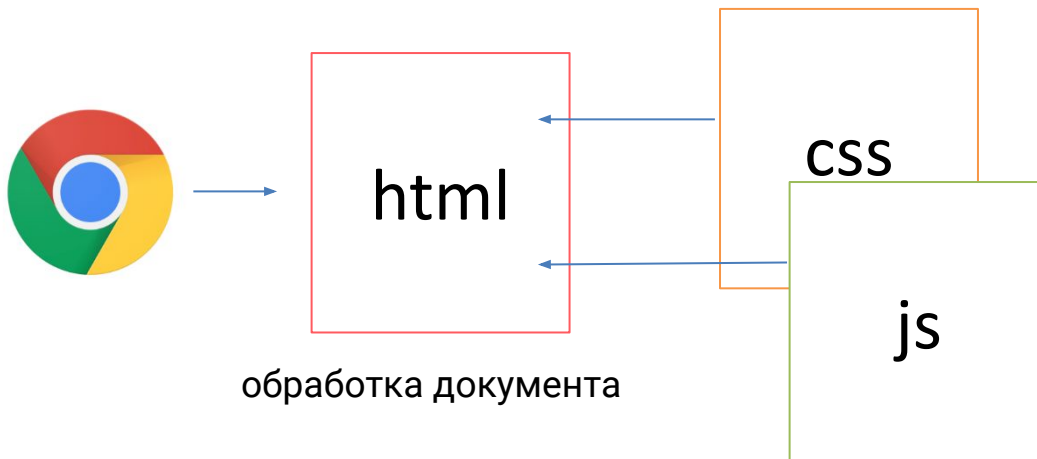
Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



Ресурсы

После того как ресурс загружен, он начинает обрабатываться браузером.

Например, браузер обрабатывал наш JS при каждой загрузке, выполняя каждую строчку с верха файла* до самого его низа:



Примечание*: это, конечно, не совсем правда, JS работает гораздо хитрее, но для простоты эта модель нам подойдёт

Сегодня мы научимся управлять этим процессом, изменяя то, в каком порядке браузер будет выполнять наши инструкции (напоминаю, мы пока учимся писать "вычислительную" часть нашей программы, но скоро перейдём к взаимодействию с интерфейсами).



Функции



Функции

Для начала, обсудим с вами функции. Чтобы понять, что это такое, давайте посмотрим на то, как в реальном мире вы вызываете такси (по шагам, так же, как мы делали до этого в наших небольших JS-программах):

1. Вы берёте телефон (наверное, при этом его ещё нужно разблокировать – пин-код или отпечаток пальца)
2. Вводите номер такси
3. Дождитесь, пока оператор ответит вам
4. Называете адрес (или ориентир), куда таксист должен подъехать

Возможны, конечно, нюансы, но в целом процесс выглядит именно так.



Функции

А теперь представьте, что вы объясняете этот процесс другу (или кому-то ещё), который никогда этого не делал (не вызывал такси).

В первый раз вы ему целиком по порядку опишите все шаги. А затем для этих всех шагов придумаете имя, например, "вызвать такси".

И в следующий раз вы уже не будете своему другу повторять все шаги, а просто скажете "вызови такси".

Давайте на миг задумаемся, почему мы делаем именно так, а не иначе? И ведь на самом деле так происходит с любым набором шагов.



Функции

Делаем мы так по одной простой причине - это удобно. Нам не приходится каждый раз повторять одни и те же 4 шага (а если бы их было 20?).

Ключевой момент: чтобы использовать это ("заказать такси", "заказать пиццу", "снять наличные в банкомате", "закинуть на счёт") нам нужно:

1. Самим знать последовательность действий
2. Объяснить эту последовательность действий другому человеку (или удостовериться, что он понимает её так же как мы).



Функции

В программировании всё точно так же: мы можем объяснить браузеру, что есть последовательность шагов (инструкций) и мы хотим дать ей (этой последовательности) имя, чтобы потом пользоваться этим удобным именем, а не повторять его много раз.

Мы будем говорить, что эта именованная последовательность – это и есть функция. Т.е. функция для нас – это "кусочек" кода, у которого есть имя*.

Примечание*: чуть позже мы расширим это определение и увидим функции, у которых не будет имён (нам лениво будет придумывать им имена).



Функции

Зачем это нужно в программировании? Ведь мы же спокойно до этого писали код, который в браузере выполнялся при загрузке страницы?



Функции

Давайте ещё раз посмотрим на одно из ваших предыдущих ДЗ:



Вы перемещаете ползунок (1), а числа (2 и 3) должны автоматически пересчитываться. Это значит, что при изменении суммы вклада каждый раз заново нужно пересчитывать значения по формуле.



Функции

А раз мы собираемся делать одно и то же каждый раз при перемещении ползунка, почему бы этому "одному и тому же" не дать удобное имя и просто им пользоваться? Так и поступим.

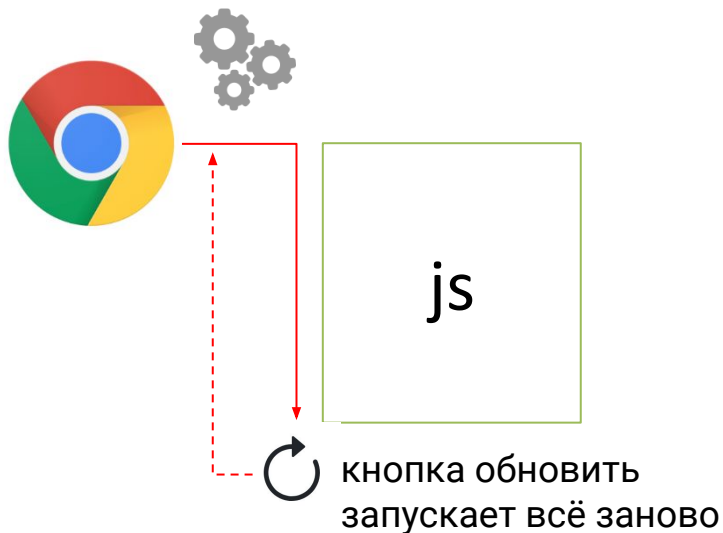
Для решения этой задачи создадим отдельный проект `deposit`, в котором и будем размещать весь наш код.

Не ленитесь и на каждую задачу создавайте отдельный проект (желательно руками, а не через `Ctrl+C`, `Ctrl+V`). Навыки вырабатываются только через повторение, а не через копирование.



Функции

Естественно у вас должен возникнуть вопрос, как такое сделать (заставить выполняться код снова и снова), если браузер обрабатывает весь наш файл JS при его загрузке?



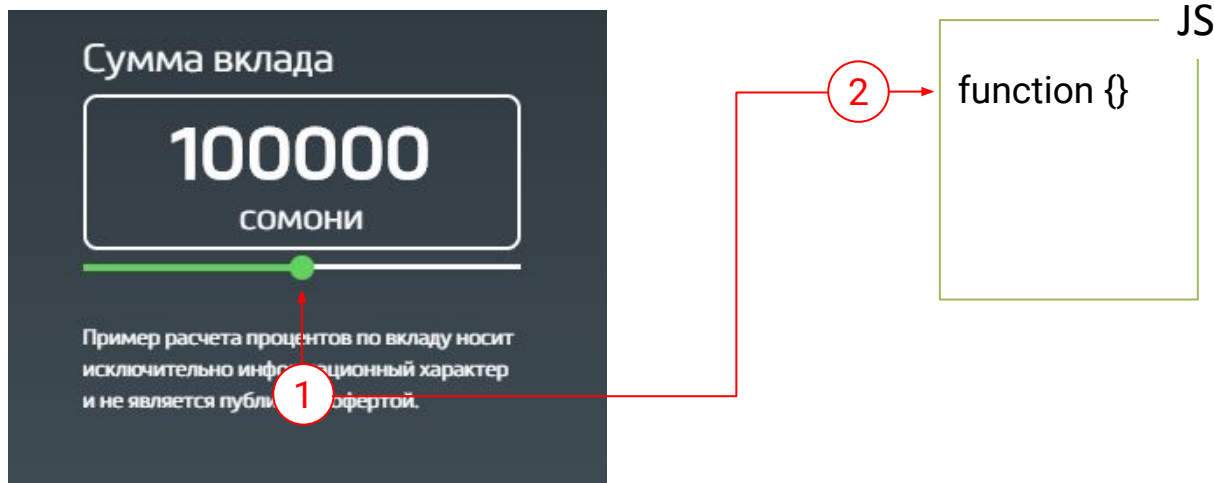
Можно, конечно, заставить пользователя при изменении значения ползунка "перезагружать" страницу, но он врядли обрадуется этому – это устаревший подход (его ещё иногда называют "олдскульным").

Так делали тогда, когда возможности JS были ограничены (например, писали сервис целиком на PHP). Сейчас так делать не стоит.



Функции

Вернёмся к нашей задаче. Что же мы можем сделать? Мы можем "попросить" браузер при каждом изменении значения ползунка выполнять кусочек кода. В качестве этого кусочка кода вполне подойдёт тот, которому мы дали имя (наша функция):

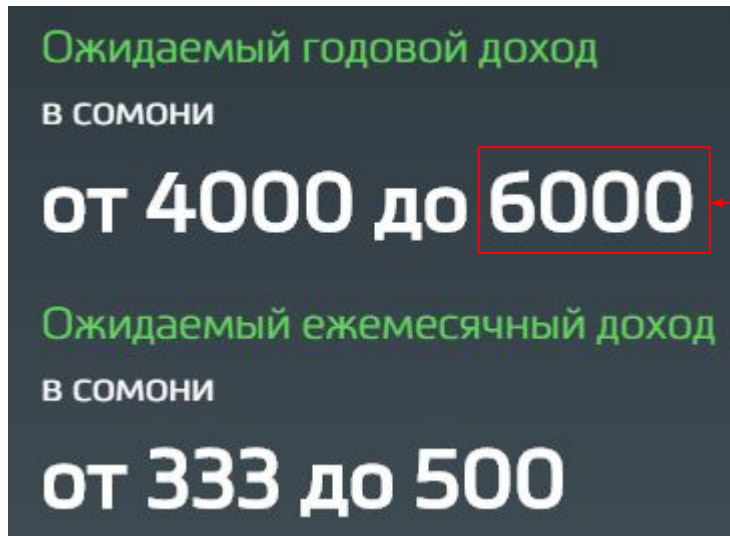


Вы перемещаете ползунок (1), а браузер вызывает функцию (2), которая всё пересчитывает. Фактически, теперь вы уже знаете, как "всё устроено". Остаётся только научиться этим пользоваться.



Функции

Для простоты мы напишем функцию, которая будет считать только максимальный годовой доход:



Без функции

Без функции наша программа бы выглядела вот так:

```
JS app.js  X
js > JS app.js > ...
1  const input = 10000;
2  const maxPercent = 0.06;
3
4  const maxAmount = input * maxPercent;
5
6  console.log(maxAmount);
```

Сместим весь код вниз на две строки и начнём набирать слово "function":

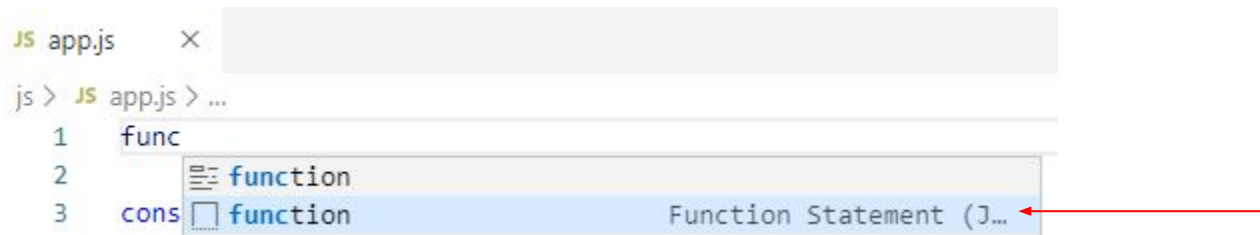
```
JS app.js  X
js > JS app.js > ...
1  fu
2  function
3  co function
   Function Statement (J...
```

Появилось "автодополнение", которое за вас допишет код. Если вы просто нажмёте клавишу **Tab** (как в Emmet), то за вас допишется только слово "function". А если выберете второй вариант (с помощью стрелки вниз на клавиатуре), то за вас допишут целую конструкцию.



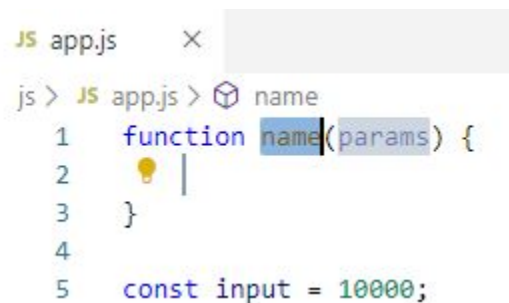
Snippet

Выберите второй вариант (если что-то пошло не так, просто нажмите **Ctrl+Z** несколько раз):



```
JS app.js ×
js > JS app.js > ...
1 func
2 function
3 cons function Function Statement (J...
```

Вы получите (это называется snippet – заготовка кода):

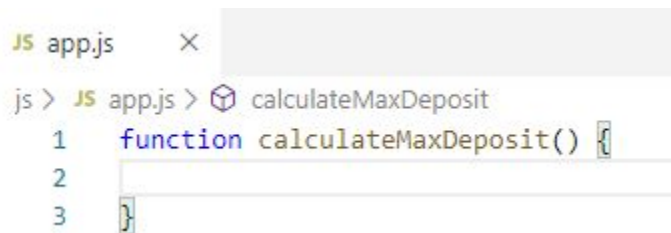


```
JS app.js ×
js > JS app.js > name
1 function name(params) {
2   |
3 }
4
5 const input = 10000;
```



Snippet

Логика работы такая же, как в Emmet: перемещаемся с помощью **Tab** по выделенным областям, изменяя их при необходимости (жмите **Ctrl+Z** для отмены и пробуйте снова, если что-то не получилось):



The screenshot shows a code editor with a tab labeled 'JS app.js'. Below the tab, the command 'js > JS app.js > calculateMaxDeposit' is entered. The editor displays a function definition with line numbers 1, 2, and 3. Line 1 contains 'function calculateMaxDeposit() {' with a cursor at the opening brace. Line 2 is empty. Line 3 contains '}' with a cursor at the closing brace.

```
JS app.js  ×  
js > JS app.js > calculateMaxDeposit  
1  function calculateMaxDeposit() {  
2  
3  }
```



Snippet

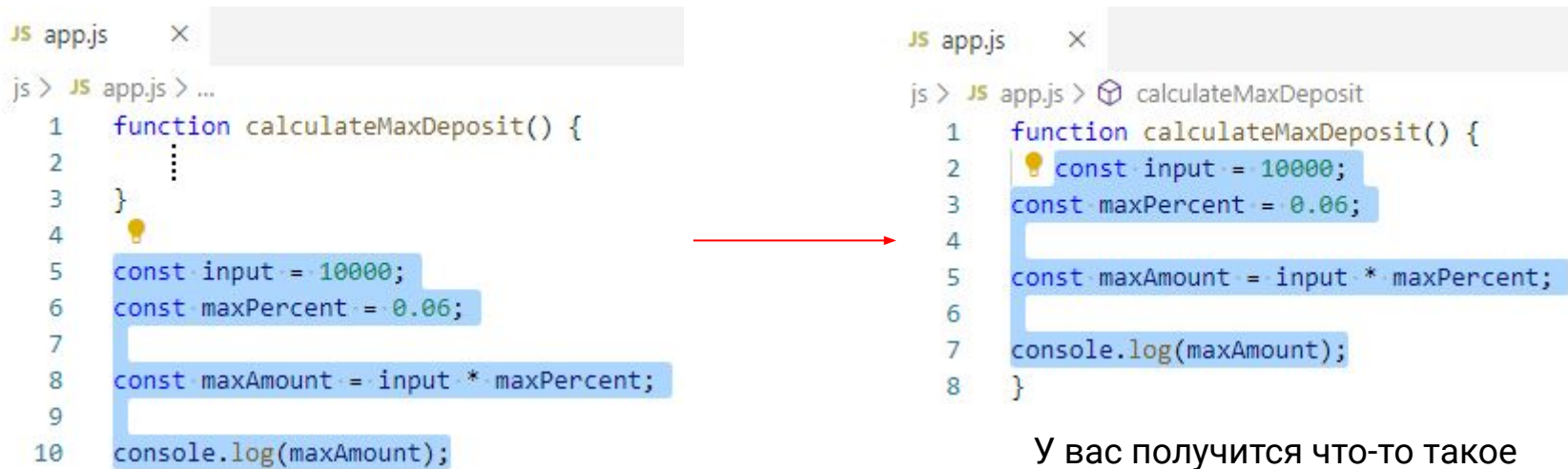
Важно: учитесь по максимуму использовать горячие клавиши и автодополнение!

Никогда не пишите весь код руками: чем меньше вы пишете руками, тем меньше ошибок допускаете.



Переносим код в функции

Дальше выполним следующую операцию: выделим весь код (кроме того, что нам сгенерировал VS Code) и "перетянем" его в позицию между фигурными скобками:



```
JS app.js  x
js > JS app.js > ...
1  function calculateMaxDeposit() {
2      ...
3  }
4  ⚡
5  const input = 10000;
6  const maxPercent = 0.06;
7
8  const maxAmount = input * maxPercent;
9
10 console.log(maxAmount);
```

→

```
JS app.js  x
js > JS app.js > calculateMaxDeposit
1  function calculateMaxDeposit() {
2      ⚡ const input = 10000;
3      const maxPercent = 0.06;
4
5      const maxAmount = input * maxPercent;
6
7      console.log(maxAmount);
8  }
```

У вас получится что-то такое

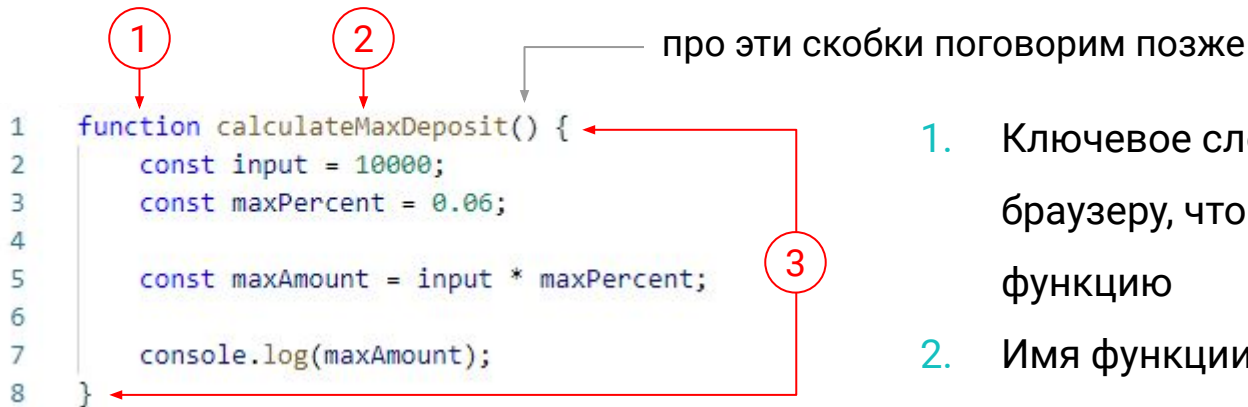
Нажмите сочетание клавиш **Alt+Shift+F**: ваш код "выровняется" и станет красивым.

Важно: всегда выравнивайте код! Представьте, что ваш код – это ваша одежда. Он должен быть аккуратный и чистый. Именно по вашему коду вас будут оценивать как программиста.



Разбираемся

Давайте разбираться с тем, что мы написали:



1. `function` — ключевое слово, которое сообщает браузеру, что мы собираемся объявить функцию

2. `calculateMaxDeposit()` — имя функции, по которому потом к ней можно будет обратиться (воспринимайте как имя переменной)

3. `{ ... }` — тело функции: та последовательность инструкций, которая будет выполняться

про эти скобки поговорим позже

```
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.06;  
4  
5   const maxAmount = input * maxPercent;  
6  
7   console.log(maxAmount);  
8 }
```

1. Ключевое слово, которое сообщает браузеру, что мы собираемся объявить функцию
2. Имя функции, по которому потом к ней можно будет обратиться (воспринимайте как имя переменной)
3. Тело функции: та последовательность инструкций, которая будет выполняться

Очень важно не напортачить с фигурными скобками! Если есть открывающая `{`, то должна быть закрывающая `}` (они всегда идут парами).



Объявление функции

Общий формат выглядит вот так:

```
function имяФункции() {  
    тело функции (может быть много строк);  
}
```

Будьте очень внимательны к мелочам! Как программист, вы должны привыкнуть, что все скобки, пробелы, точки с запятой и другие символы ставятся не просто так – они ставятся в строго отведённые места.

И если вы их поставите не туда, то браузер вас не поймёт, а значит либо не исполнит ваш код совсем, либо исполнит его не правильно (это не HTML, в JS браузер почти не прощает ошибок).



Как это работает

Если мы сейчас посмотрим в консоль (надеемся, что за неделю вы не забыли, как туда смотреть) – там будет пусто.

Что это значит? На самом деле, браузер, обрабатывает наш код по-прежнему сверху вниз, но внутрь функции (внутри фигурных скобок) он не заходит – он просто запоминает, что мы ему сообщили о том, что существует функция с конкретным именем.



Как это работает

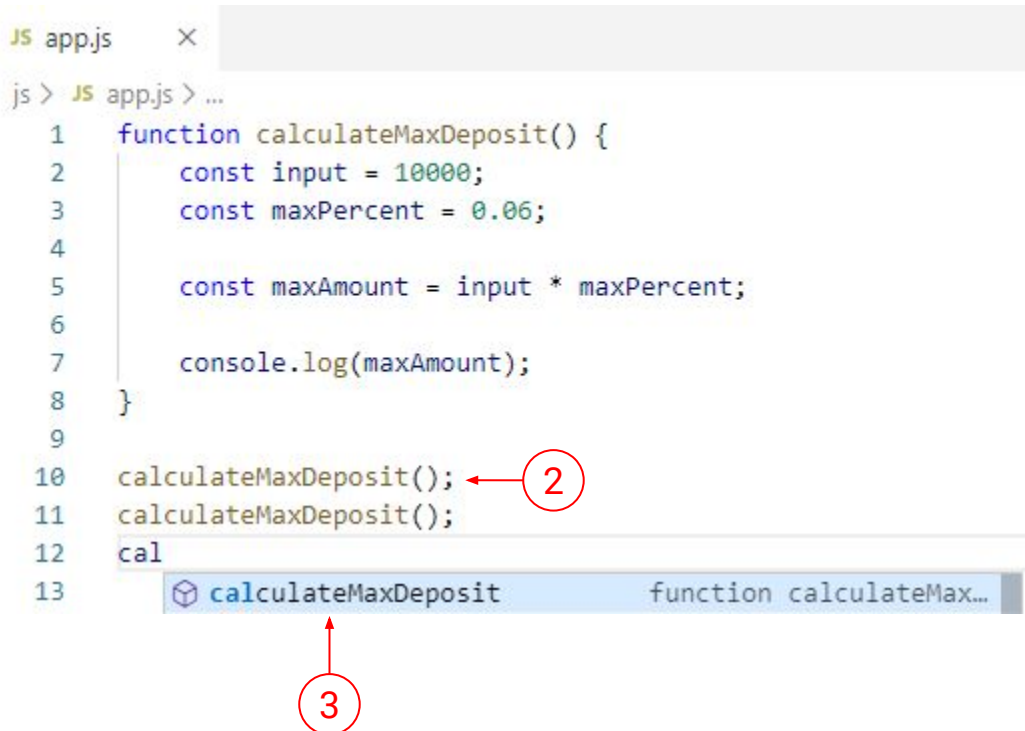
Это как в жизни: в вашем телефоне есть функция "позвонить". Т.е. вы набираете номер, нажимаете на кнопку и телефон звонит. Сам по себе телефон не звонит, пока вы эту функцию не вызовете (иначе вы бы сказали, что это "сломанный телефон").

Т.е. нам нужно научиться вызывать эту функцию.



Вызов функции

Для того, чтобы вызвать функцию, нужно написать имя функции, после чего поставить круглые скобки:



The screenshot shows a Node.js REPL session with the following code and annotations:

```
js > JS app.js > ...  
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.06;  
4  
5   const maxAmount = input * maxPercent;  
6  
7   console.log(maxAmount);  
8 }  
9  
10 calculateMaxDeposit();  
11 calculateMaxDeposit();  
12 cal  
13
```

Annotations:

- A red vertical line with a circle containing the number 1 is positioned to the left of lines 1 through 8, indicating the function definition.
- A red circle containing the number 2 has an arrow pointing to the first call to `calculateMaxDeposit();` on line 10.
- A red circle containing the number 3 has an arrow pointing to the autocomplete dropdown menu on line 13, which shows `calculateMaxDeposit` as a suggestion.

1. Объявление функции
2. Вызов функции
3. Автодополнение (используйте)



Вызов функции

Теперь, если вы три раза напишите "`calculateMaxDeposit();`" в консоль три раза выведется одно и то же число:

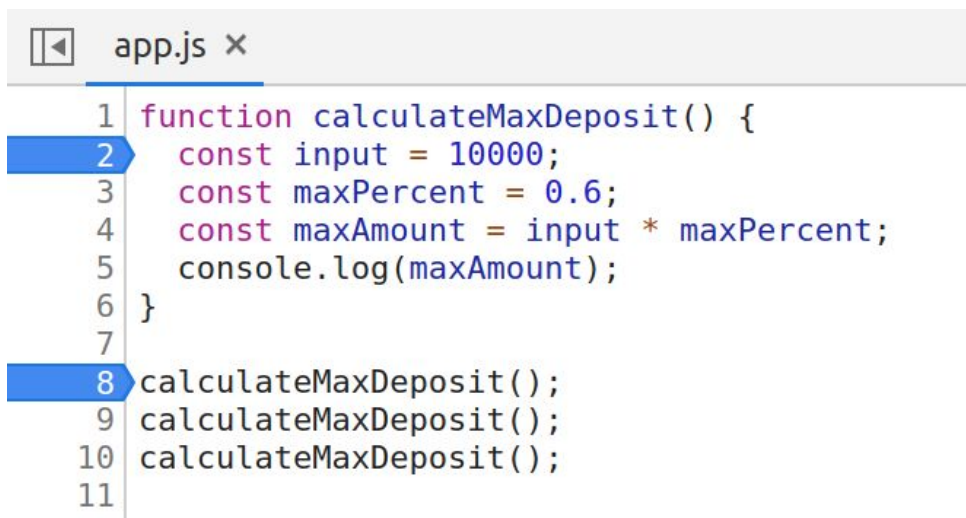


Число 3 означает, что вы печатаете три раза одно и то же, с помощью `console.log` (браузер по умолчанию группирует одинаковый вывод).



Debugging

Если мы попробуем поставить точку остановки на первой строке, то у нас не получится и браузер поставит её на вторую. Это не совсем то, что нам нужно, поэтому мы схитрим и поставим целых две точки остановки:

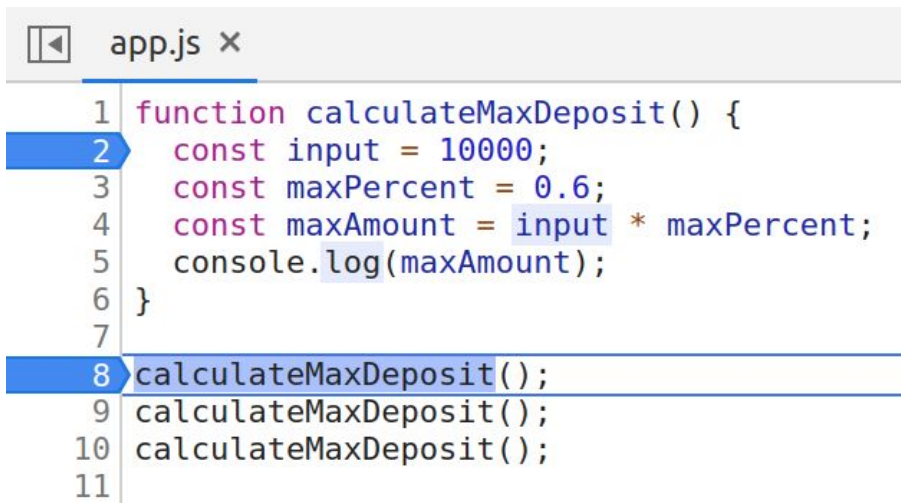


```
app.js x
1 function calculateMaxDeposit() {
2   const input = 10000;
3   const maxPercent = 0.6;
4   const maxAmount = input * maxPercent;
5   console.log(maxAmount);
6 }
7
8 calculateMaxDeposit();
9 calculateMaxDeposit();
10 calculateMaxDeposit();
11
```



Debugging

Обновим страницу и увидим, что действительно, браузер "не зашёл" внутрь функции, а остановился на первом вызове:



```
app.js x
1 function calculateMaxDeposit() {
2   const input = 10000;
3   const maxPercent = 0.6;
4   const maxAmount = input * maxPercent;
5   console.log(maxAmount);
6 }
7
8 calculateMaxDeposit();
9 calculateMaxDeposit();
10 calculateMaxDeposit();
11
```

Это значит, что браузер "прочитал" объявление функции и двинулся дальше. И следующей строкой он будет эту функцию вызывать.






Debugging

Всегда обязательно проверяйте, как работает то, что вы где-то читали/видели, в современных браузерах: информация быстро устаревает, а иногда авторы просто "ошибаются" либо не знают то, о чём пишут (конечно же, наши материалы вы тоже должны проверять таким же образом – мы специально для вас заложили ряд "ловушек", в которых вы должны разобраться).

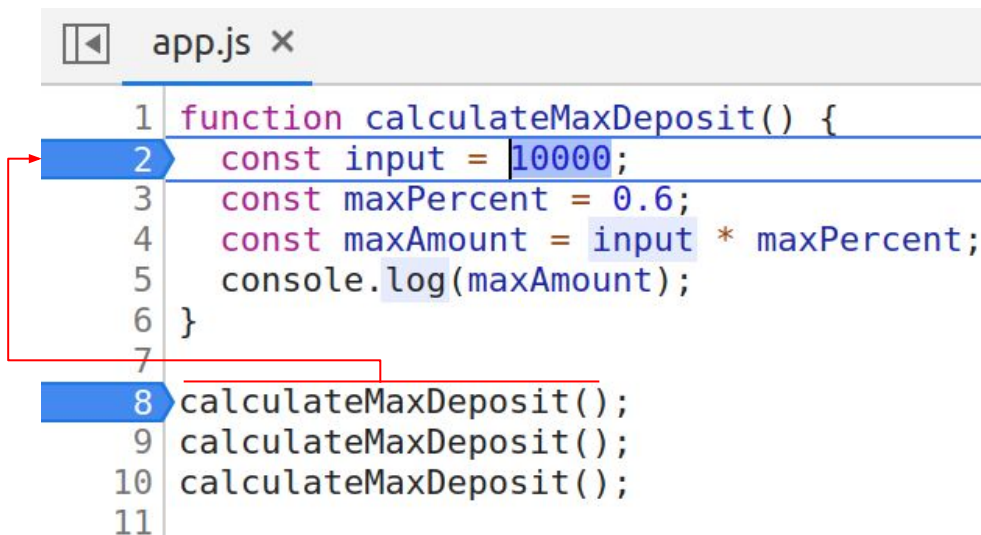


Debugging

У нас есть три варианта:

-  продолжить до следующей точки останова
-  "перешагнуть" вызов функции (сразу перейти к 11 строке, если по пути нет точек останова)
-  зайти внутрь функции

На самом деле, в текущем сценарии все три кнопки приведут нас в одно место:

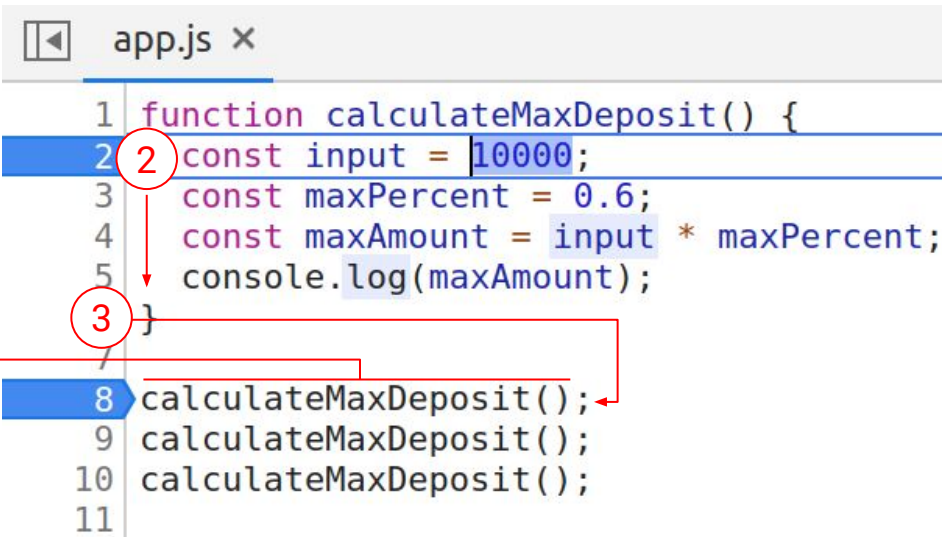


```
app.js x
1 function calculateMaxDeposit() {
2   const input = 10000;
3   const maxPercent = 0.6;
4   const maxAmount = input * maxPercent;
5   console.log(maxAmount);
6 }
7
8 calculateMaxDeposit();
9 calculateMaxDeposit();
10 calculateMaxDeposit();
11
```



Debugging

Это очень важно понять: каждый раз, когда браузер встречает вызов функции (1), он переходит в эту функцию для исполнения того кода, что в ней (функции) содержится. После того, как код функции будет исполнен (2), браузер вернётся обратно в ту точку, из которой перешёл в функцию (3):



The screenshot shows a code editor with a file named `app.js`. The code contains a function `calculateMaxDeposit()` and three calls to it. Red arrows and numbers illustrate the execution flow:

```
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.6;  
4   const maxAmount = input * maxPercent;  
5   console.log(maxAmount);  
6 }  
7  
8 calculateMaxDeposit();  
9 calculateMaxDeposit();  
10 calculateMaxDeposit();  
11
```

Annotation 1: A red circle with the number 1 is placed next to the first call `calculateMaxDeposit();` on line 8. A red arrow points from this circle to the function definition on line 1.

Annotation 2: A red circle with the number 2 is placed next to the first line of the function body, `const input = 10000;` on line 2. A red arrow points from this circle to the closing brace of the function on line 6.

Annotation 3: A red circle with the number 3 is placed next to the closing brace of the function on line 6. A red arrow points from this circle back to the first call on line 8, indicating the return path.



Debugging

Вначале этом может показаться сложным, но попробуйте в дебаггере пройти это раза 3-4, после этого вы поймёте, как всё происходит.

Важно: обязательно сделайте это упражнение, без понимания того, как вызываются функции и что происходит, дальше двигаться бессмысленно.



const

Теперь интересный вопрос: мы с говорили, что с помощью `const` имя можно объявить всего один раз? А тут получается мы вызываем функцию и код, который содержит объявление `const` выполняется целых три раза без ошибок. Как так?



Области видимости



console.log

`{ }` создают область видимости для переменных, объявленных с помощью `let` и `const`.

Что такое область видимости? Это участок кода, внутри которого это имя существует. Оно (имя) "резервируется" при заходе в этот участок кода и освобождается при выходе из него:

The screenshot shows a code editor with a file named `app.js`. The code is as follows:

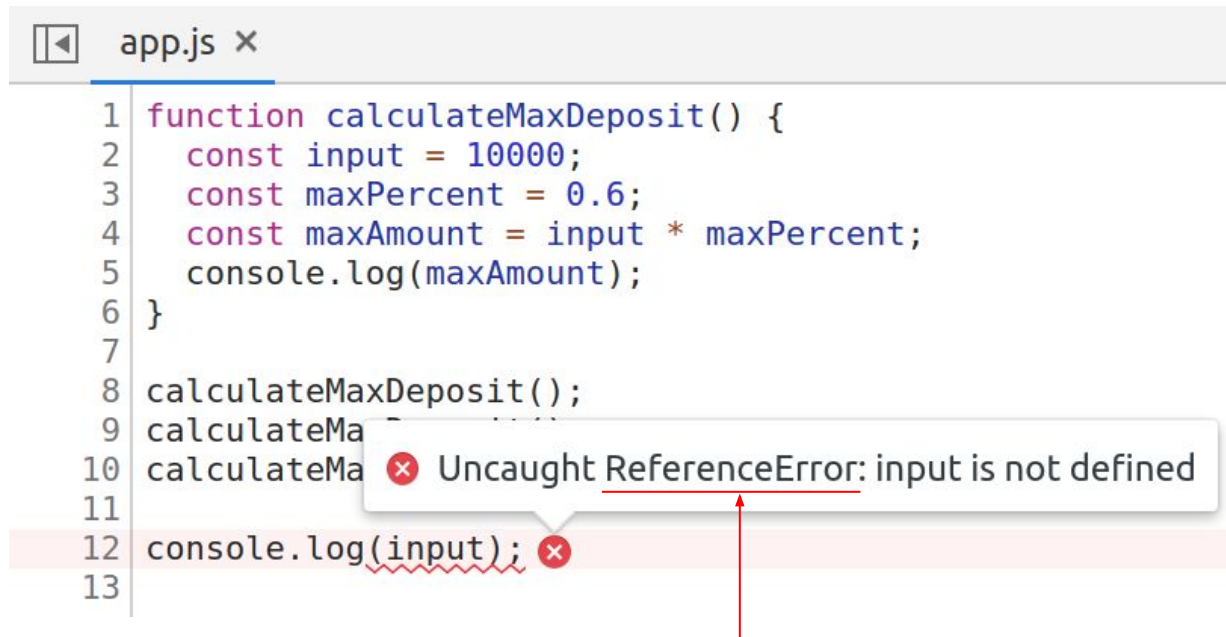
```
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.6;  
4   const maxAmount = input * maxPercent;  
5   console.log(maxAmount);  
6 }  
7  
8 calculateMaxDeposit();  
9 calculateMa  
10 calculateMa  
11  
12 console.log(input);  
13
```

A red bracket on the right side of the code, spanning from line 2 to line 5, groups the declarations of `input`, `maxPercent`, and `maxAmount`. Next to this bracket is the text: `input, maxPercent, maxAmount` существуют **только внутри функции**.

An error message is displayed in a tooltip over line 12: `Uncaught ReferenceError: input is not defined`. A red arrow points from the text "только внутри функции" to the error message, indicating that the variable `input` is only available within the function's scope and is not accessible in the global scope where line 12 is executed.



Области видимости



The screenshot shows a code editor with a file named 'app.js'. The code defines a function 'calculateMaxDeposit' with three local constants: 'input' (10000), 'maxPercent' (0.6), and 'maxAmount' (input * maxPercent). The function logs 'maxAmount' and is called on line 8. On line 10, the function is called again, but on line 12, it attempts to log 'input'. A red squiggly line under 'input' on line 12 indicates an error. A tooltip points to this error with the message 'Uncaught ReferenceError: input is not defined'. A red arrow points from the text below to the error message.

```
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.6;  
4   const maxAmount = input * maxPercent;  
5   console.log(maxAmount);  
6 }  
7  
8 calculateMaxDeposit();  
9 calculateMa  
10 calculateMa  
11  
12 console.log(input);  
13
```

Uncaught ReferenceError: input is not defined

запомните эту ошибку: она означает, что вы пытаетесь обратиться к имени, которого в этой области нет



Блок кода

`{}` создают блок кода, а `let` и `const` создают имена, обладающие блочной областью видимости (т.е. не выходящие за пределы блока).

Запомнить это достаточно просто: если вы видите, что какой-то кусок кода находится внутри `{}` – это и есть блок. Любые переменные, объявленные с помощью `let` и `const` не могут "выбраться" из этого блока (всё, что объявлено в блоке – остаётся в блоке) и становятся недоступны после выхода из блока.

В JS встроен механизм сборки мусора, который следит за тем, что вы не используете, и аккуратно удаляет, чтобы не нагружать компьютер пользователя. Поэтому не бойтесь создавать переменные – это очень дёшево и позволит вам писать хороший код.



TDZ advanced

Это материал повышенной сложности (помечен значком ^{advanced}): он не обязателен для прохождения курса, но его часто спрашивают на собеседованиях, поэтому мы его приводим, чтобы вы (в оставшееся от ДЗ время) могли с ним ознакомиться.

Но мы же говорили про переменные, что они появляются только тогда, когда браузер выполнит строку с объявлением? Почему мы сейчас говорим, что имя резервируется при входе в блок?



TDZ advanced

На самом деле браузер работает немного хитрее, чем мы рассматривали на предыдущих лекциях: зайдя в блок он, фактически, резервирует все имена `let` и `const`, но обратиться к ним нельзя до тех пор, пока они не встретятся сами строки с объявлением (с `let` или `const`):

```
1 function calculateMaxDeposit() {  
2   const input = 10000;  
3   const maxPercent = 0.6;  
4   const maxAmount = input * maxPercent;  
5   console.log(maxAmount);  
6 }
```

TDZ: `maxAmount` зарезервировано, но станет доступно для использования только после выполнения 5-ой строки, если обратиться раньше – будет `ReferenceError`

Это называется "временно мёртвая зона" (Temporal Dead Zone), т.е. имя уже существует, но использовать его пока нельзя*.

Примечание*: здесь есть одна хитрость, которая похожа на исключение из этого правила (что не так), которую мы рассмотрим, когда будем проходить замыкания.



TDZ advanced

Как это запомнить? Представьте, что вы пришли в кафе и заняли место для себя и своего друга (но он ещё не подошёл – придёт чуть позже).

С одной стороны: место занято, с другой стороны: на вас будут очень странно смотреть, если вы будете разговаривать с человеком, который ещё не пришёл.



Входные параметры и результат



Параметры и результат

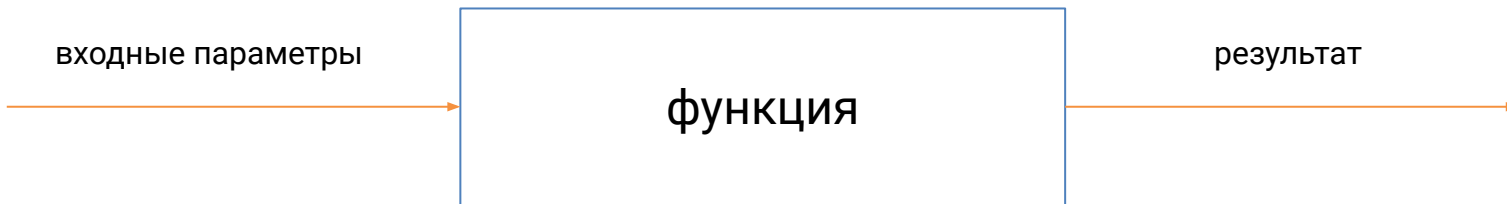
Это всё хорошо, но не особо осмысленно, ведь мы хотели, чтобы результат каждый раз менялся, если пользователь двигает ползунок: сейчас же результат один и тот же (сколько нашу функцию не вызывай).

Один из вариантов решить эту проблему (и самый лучший) – сделать так, чтобы функция при вызове могла принимать какие-то данные, а в ответ возвращать какой-то результат.



Параметры и результат

Это как в нашей с вами системе проверки ДЗ: вы загружаете архив с ДЗ, в ответ получаете результат (OK или FAIL):



Параметры

Для начала, нам нужно определиться с параметрами (это то, что мы подаём на вход функции). Например, оплачивая телефон, вы вводите номер телефона и сумму.

В нашем случае, входным параметром будет только сумма, а процент депозита будет фиксирован. Почему мы решили именно так, а не иначе? На самом деле, правильного ответа тут нет и процент тоже можно было сделать входным параметром. В этом и заключается сложность программирования: нет единственно правильного способа что-то сделать.



Параметры

Мы с вами пойдём по простому пути и будем выносить в параметры только то, что точно меняется. Этот простой приём позволит вам писать код, а не бесконечно размышлять "как же будет лучше" (эти размышления иногда называют "паралич анализа" – вы слишком долго размышляете и не можете решить, как же лучше).



Параметры

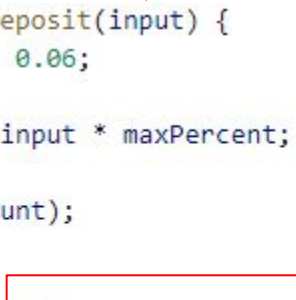
Параметры пишутся в круглых скобках сразу за именем функции (в объявлении и вызове):

было

```
1 function calculateMaxDeposit() {  
2     const input = 10000;  
3     const maxPercent = 0.06;  
4  
5     const maxAmount = input * maxPercent;  
6  
7     console.log(maxAmount);  
8 }  
9  
10 calculateMaxDeposit();  
11 calculateMaxDeposit();  
12 calculateMaxDeposit();
```

стало

```
1 function calculateMaxDeposit(input) {  
2     const maxPercent = 0.06;  
3  
4     const maxAmount = input * maxPercent;  
5  
6     console.log(maxAmount);  
7 }  
8  
9 calculateMaxDeposit(10000);  
10 calculateMaxDeposit(1000);  
11 calculateMaxDeposit(100);
```



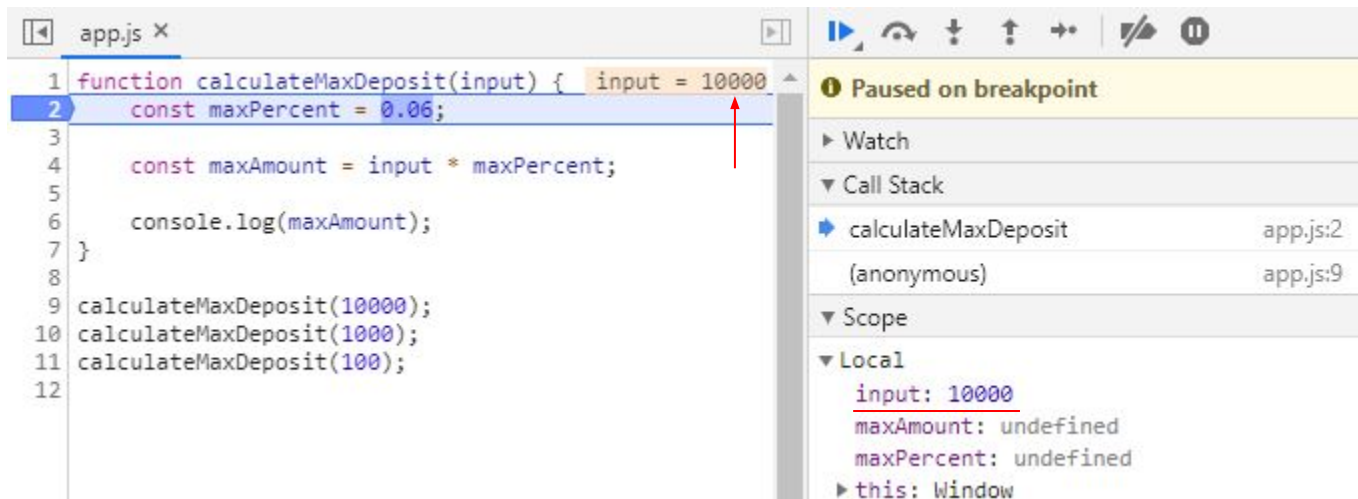
В объявлении вы даёте параметру имя (**input**), а в вызове кладёте в это имя конкретное значение (**10000**, **1000**, **100**).

Для простоты вы можете представлять, что параметр **input** – это просто переменная, объявленная с помощью **let** (не **const**), в которую браузер вам "кладёт" значения (при каждом вызове).

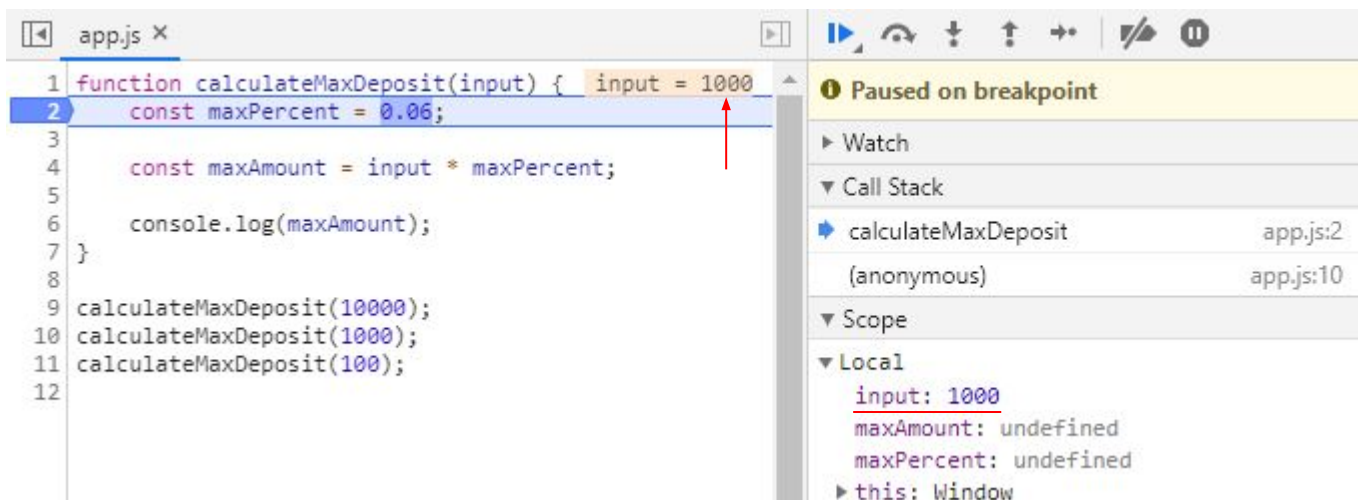


Debugger

Для вызова на 9-ой строке:

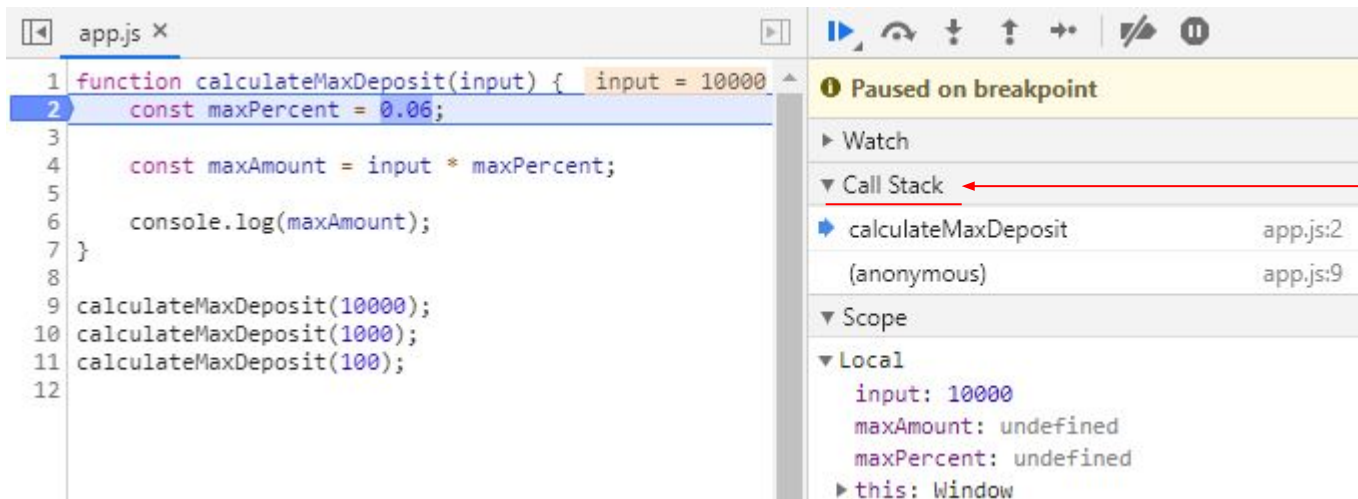


Для вызова на 10-ой строке:



Call Stack

Обратите внимание на боковую панельку Call Stack:



В ней в обратном порядке указано, откуда вы попали в текущую точку: из `app.js` (строка 9) мы попали в функцию `calculateMaxDeposit app.js` (строка 2).

Q: что значит в обратном порядке?

A: представьте, что вы поднимаетесь по лестнице – здесь также, самый первый шаг – вниз.



Параметры vs Аргументы

Иногда вы можете встретить термин аргумент – а именно конкретное значение передаваемое в функцию:

```
1  function calculateMaxDeposit(input) {  
2      const maxPercent = 0.06;  
3  
4      const maxAmount = input * maxPercent;  
5  
6      console.log(maxAmount);  
7  }  
8  
9  calculateMaxDeposit(10000);  
10 calculateMaxDeposit(1000);  
11 calculateMaxDeposit(100);
```

параметр

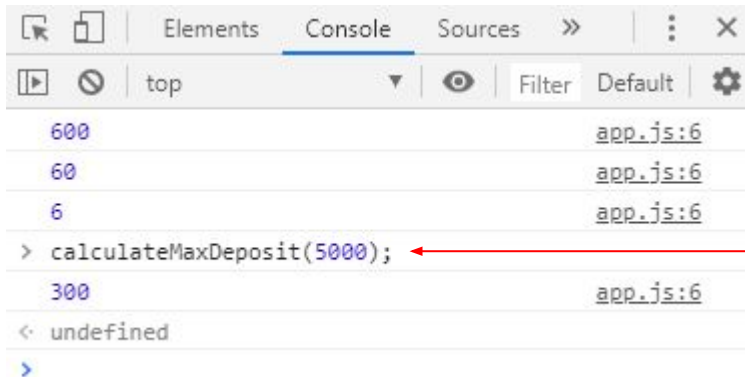
аргумент (или текущее значение параметра в рамках этого конкретного вызова)

Для простоты мы везде будем употреблять термин параметр.



Параметры

Теперь наша функция стала удобнее – она умеет считать депозит в зависимости от суммы:



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following log entries:

- 600 (app.js:6)
- 60 (app.js:6)
- 6 (app.js:6)
- > calculateMaxDeposit(5000); (app.js:6) ← A red arrow points from this line to the text below.
- 300 (app.js:6)
- < undefined

И мы даже можем вызвать её с нужными параметрами прямо из консоли. Но вы должны запомнить, что печатать что-то в консоль из функции – это не очень хорошая идея.

Но что это за `undefined`? И почему печатать в консоль – плохо?



Возвращаемое значение

В JS все функции возвращают значение. Что это значит? Это значит на то место, где стоял вызов функции подставляется результат выполнения этой функции. Пока ничего не понятно, но давайте пойдём по шагам.

Вспомним одну из первых лекций:

JS app.js



js > JS app.js

1 10000 / 0.147

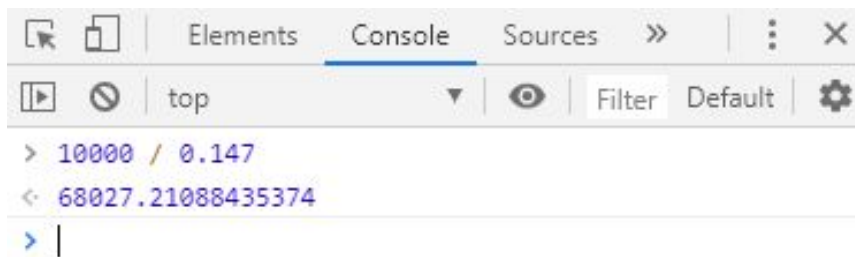
2

есть выражение, которое вычисляется, но

поскольку результат никак не используется, то он

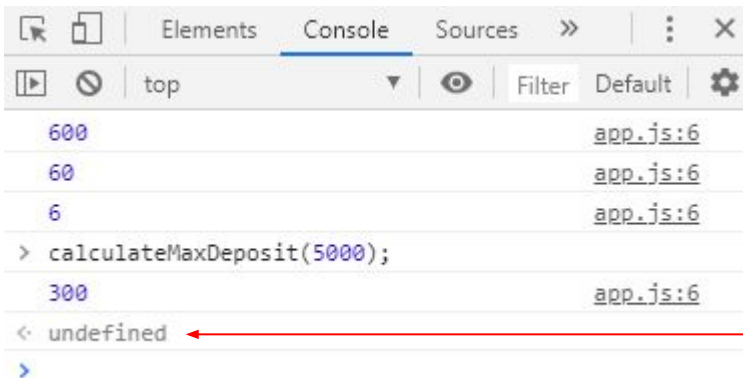
(результат) полностью игнорируется.

Но если вы то же самое пишете в консоли, то результат не игнорируется, а выводится вам:



Возвращаемое значение

Таким образом, на то место, где стоял вызов функции подставляется `undefined`:



```
600 app.js:6
60 app.js:6
6 app.js:6
> calculateMaxDeposit(5000);
300 app.js:6
< undefined
```

Обратите внимание: консоль любит использовать цветовую индикацию для разных типов данных, например, числа выводятся синим, а `undefined` – серым.

`undefined` значит буквально "не определено". Чуть позже мы его детально разберём, пока же научимся возвращать из функции значение.



Результат

Для того, чтобы что-то вернуть из функции, необходимо использовать ключевое слово `return`:

```
app.js x
1 function calculateMaxDeposit(input) {
2   const maxPercent = 0.6;
3
4   const maxAmount = input * maxPercent;
5
6   return maxAmount;
7 }
8
9 const result = calculateMaxDeposit(1000);
10
11 console.log(result);
```



Результат

```
app.js x
1 function calculateMaxDeposit(input) {
2   const maxPercent = 0.6;
3
4   const maxAmount = input * maxPercent;
5
6   return maxAmount;
7 }
8
9 const result = calculateMaxDeposit(1000);
10
11 console.log(result);
```

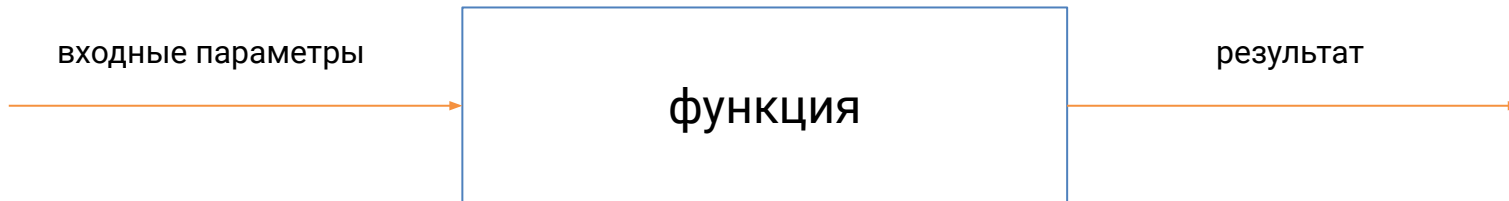
Как это работает? На 9-ой строке стоит оператор `=`. Оператор `=` работает справа-налево, т. е. сначала вычисляется всё, что стоит с правой части (1), затем это присваивается имени слева (3).

В правой части стоит вызов функции, это значит, что браузер выполнит всё, что находится внутри функции и на место вызова этой функции подставит то, что стоит после `return` (2).



Параметры и результат

Таким образом, мы добиваемся схемы, к которой стремились:



Мы разобрались с тем, как объявлять функции, передавать в них параметры, возвращать из функции результат.

Важно: такие функции, которые на основании своих входных параметров рассчитывают и возвращают результат (при этом не взаимодействуют с внешним миром, т.е. ничего никуда не печатают, не вызывают `alert`'ов, не меняют ничего на веб-странице) называют чистыми функциями (pure functions).

Для любых вычислительных задач старайтесь писать именно чистые функции.




Функции

У вас может возникнуть вопрос: почему мы всегда писали объявление функции наверху, а вызов только после? По-другому работать не будет?

На самом деле – будет, потому что браузер, обрабатывая JS-файл, собирает все объявления функций и "вытаскивает" их наверх (как будто они всегда там были), поэтому для всего остального кода они уже существуют:

```
1  const result = calculateMaxDeposit(1000);  
2  
3  console.log(result);  
4  
5  function calculateMaxDeposit(input) {  
6      const maxPercent = 0.06;  
7  
8      const maxAmount = input * maxPercent;  
9  
10     return maxAmount;  
11 }
```

объявления функций "всплывают" наверх



Этот код тоже будет работать.



undefined



undefined

`undefined` это такая интересная конструкция JS: с одной стороны это тип данных, с другой стороны – это значение (зависит от того, в каком смысле его употребляют).

`undefined` встречается тогда, когда вы:

1. Создали переменную, но не присвоили ей значение (через `let/var`, но не `const`)
2. Не передали параметр в функцию
3. Не вернули ничего из функции, но пытаетесь этим воспользоваться
4. Пытаетесь обратиться к несуществующему свойству объекта (пройдём на следующей лекции)

В любом случае, если вы встречаете `undefined`, то на первых порах – это скорее всего ваша ошибка (вы что-то где-то не доглядели).



undefined

По-житейски, вы можете воспринимать `undefined` как "Не знаю".

Например, вы у продавца в магазине спрашиваете, сколько стоит бутылка воды, а он вам говорит "не знаю". А вы хотите купить две. Сколько вы ему должны заплатить денег?

```
> undefined * 2  
< NaN
```

* приводит `undefined` к числу `NaN`, а `NaN * 2` будет снова `NaN`

Периодически, вы будете наталкиваться на `undefined` на сайтах, если их разработчики не продумали хорошо код на JS. Поэтому нам надо детально разобраться с `undefined`.



Переменная без значения

```
let count;  
console.log(count);
```

В консоль будет выведено `undefined`. Несмотря на то, что в `count`, скорее всего, должно быть число. Как вы помните из предыдущего слайда, почти все арифметические операции с `undefined` будут возвращать `NaN`, поскольку `undefined` приводится к числу `NaN` (и ничего хорошего в этом нет).

```
> undefined / 0  
< NaN  
  
> undefined * 0  
< NaN  
  
> undefined + 0  
< NaN  
  
> undefined - 0  
< NaN  
  
> undefined ** 0  
< 1
```

чтобы не запоминать это, не используйте в одном выражении разные типы данных



Переменная без значения

С `const` такой проблемы нет, поскольку `const` вы должны сразу инициализировать (т.е. привязать к имени значение):

```
const count;  
console.log(count);
```

Сразу получим ошибку в консоли:

```
✖ Uncaught SyntaxError: Missing initializer in const declaration
```

```
>
```

Поэтому везде старайтесь использовать `const` (если вы, конечно, не фанат `undefined`).



Не передали параметр

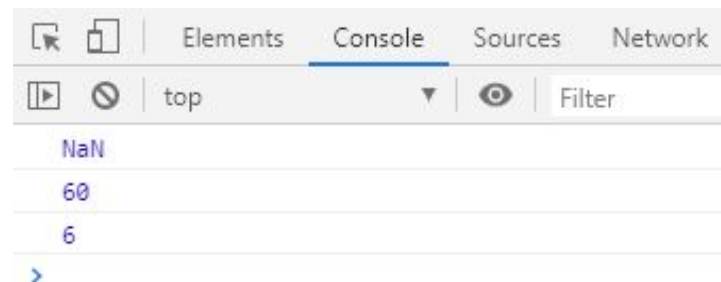
JS очень гибкий и мощный язык. И иногда он считает, что вы знаете, что делаете (и не оставит вас от необдуманных поступков).

Например, вы можете объявить функцию с одним параметром, но передавать при вызове можете:

1. Ни одного
2. Ровно один
3. Больше одного

Поставьте точку остановки на 2-ой строки и убедитесь в этом.

```
1 function calculateMaxDeposit(input) {  
2     const maxPercent = 0.06;  
3  
4     const maxAmount = input * maxPercent;  
5  
6     console.log(maxAmount);  
7 }  
8  
9 calculateMaxDeposit(); // нет параметров  
10 calculateMaxDeposit(1000); // один параметр  
11 calculateMaxDeposit(100, 10, 1); // больше одного
```

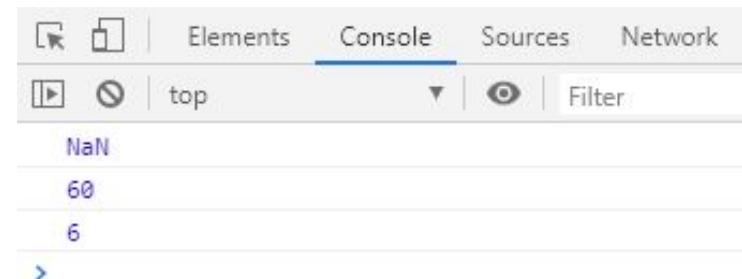


Не передали параметр

Что происходит?

1. Если мы не передаём параметр, то браузер говорит "я не знаю" (**undefined**) и кладёт это значение в **input**, поэтому результат будет **NaN**
2. Если передаём один параметр, то браузер использует его
3. Если передаём много параметров, то браузер берёт ровно столько, сколько нужно, остальные запасая в секретном месте (о нём поговорим позже)

```
1 function calculateMaxDeposit(input) {  
2     const maxPercent = 0.06;  
3  
4     const maxAmount = input * maxPercent;  
5  
6     console.log(maxAmount);  
7 }  
8  
9 calculateMaxDeposit(); // нет параметров  
10 calculateMaxDeposit(1000); // один параметр  
11 calculateMaxDeposit(100, 10, 1); // больше одного
```



Не вернули ничего

В JS функции возвращают значение, которое указано после `return`.

Если после `return` ничего не указано, то вернётся `undefined` (как будто вы вместо `return` написали `return undefined`).

Если `return` вообще внутри функции нет, то тоже вернётся `undefined` (как будто вы в самом конце функции написали `return undefined`).

В обоих этих случаях можете считать, что браузер говорит: "я должен что-то вернуть из этой функции, но программист не сказал что именно", поэтому он возвращает "не знаю" (`undefined`).



undefined

Вы должны наизусть заучить, в каких случаях получается `undefined`. В противном случае ошибки для вас станут непреодолимой стеной и вы будете наугад менять код, в попытке понять, что и где надо исправить, чтобы "всё заработало".



Условия



Условия

Разберём следующую задачу: пусть у нас есть дебетовая карта и мы хотим пользователю выдавать информацию о том, какую комиссию (1) с него возьмут в "чужих" банкоматах (это хорошая практика, заранее уведомлять об этом пользователя):

Операции

Получение наличных в банкомате Банка	0%	
Получение наличных в банкомате других банков	1% (мин. 0,35 с.)	1
Получение наличных в банкомате по зарплатной карте**	0%	2
Получение наличных в ПВН**** других банков	до 1,5% (мин. 0,35 с.)	
Получение наличных в кассах Банка	0%	

Обратите внимание, что процент может быть разный (2).



Условия

Итак, для упрощения мы уберём зарплатных клиентов и оставим только 1%, но не меньше 0.35 сомони. Т.е. если клиент захочет снять 10 сомони, то комиссию с него возьмут не 1%, а 0.35 сомони.



Алгоритм

Общий алгоритм (алгоритм – это законченный набор шагов) будет состоять из двух шагов:

1. Считаем комиссию в 1%
2. Если комиссия меньше 0.35 сомони, то комиссия = 0.35 сомони, в противном случае комиссия = комиссии из п.1

Начнём по шагам: сначала напомним функцию, которая умеет выполнять вычисления по первому пункту:

```
function calculateCommission(input) {  
  const percent = 0.01;  
  const commission = input * percent;  
  return commission;  
}  
  
const result = calculateCommission(1000);  
console.log(result);
```

Важно: не пытайтесь сразу написать функцию целиком, идите по шагам.



Условия

Если первая часть работает и у нас успешно получилось 10 сомони, то можем двигаться дальше (если нет – то добиваемся того, чтобы получилось).

Поговорим об условиях. Условие – это такая конструкция, которая позволяет вам выполнять участки кода в зависимости от результата вычисления какого-то выражения. Звучит сложно, не правда ли? По факту, мы просто будем выполнять какой-то код, если какое-то выражение принимает определённое значение.

Например, мы будем возвращать 0.35, если размер комиссии будет меньше 0.35 сомони.



Условия

Как это выглядит в коде:

```
01 function calculateCommission(input) {  
02   const percent = 0.01;  
03   const minCommission = 0.35;   
04   let commission = input * percent;  
05   if (commission < minCommission) {  
06     commission = minCommission;  
07   }  
08   return commission;  
09 }  
10  
11 const result = calculateCommission(10);  
12 console.log(result);
```

Ввели переменную для хранения минимальной комиссии

Если комиссия меньше минимальной, берём за комиссию минимальную

`if` переводится как "если", дальше в скобках пишется условие при выполнении которого выполняется блок кода (напоминаем, блок кода – это кусок кода, ограниченный `{}`).

Т.е. в нашем случае мы выполняем строку 6 только тогда, когда комиссия меньше минимальной (и попросту заменяем итоговую комиссию минимальной).



Условия

Для выполнения сравнения чисел в JS существуют следующие операторы:

1. $a < b$ – меньше
2. $a > b$ – больше
3. $a \leq b$ – меньше или равно
4. $a \geq b$ – больше или равно
5. $a == b$ – равно (именно два `=`, т.к. один – это оператор присваивания)
6. $a != b$ – не равно

Попробуем в консоли:

```
> 10 > 10  
< false  
-----  
> 10 >= 10  
< true  
-----  
> 10 != 11  
< true
```



boolean

`true` и `false` – это специальные значения типа `boolean`. Они используются для логических операций.

Логические операции – это некоторые утверждения, которые могут быть либо истинны (`true`), либо ложны (`false`). Например, мы говорим `10 > 3` – это истина (`true`):

```
> 10 > 3  
< true
```

В реальной жизни с этим немного сложнее: вы можете спросить у человека "сегодня пойдёшь?", он может ответить: "да", "нет", "не знаю", "не очень", "да нет наверное", "наверное" и ещё тысячу разных вариантов.

В JS у нас только два – `true` (да) или `false` (нет) и это хорошо.



boolean

Соответственно, если выражение в круглых скобках принимает значение **true**, то блок кода с **if** выполняется, если принимает значение **false**, то нет. Убедитесь вы этом самостоятельно, используя дебаггер и поставив точку остановки внутри **if**:

```
1 function calculateCommission(input) { input = 10
2   const percent = 0.01; percent = 0.01
3   const minCommission = 0.35; minCommission = 0.35
4   let commission = input * percent; commission = 0.1, input = 10,
5   if (commission < minCommission) { minCommission = 0.35
6   commission = minCommission;
7 }
8 return commission;
9 }
10
11 const result = calculateCommission(10);
12 console.log(result);
```



boolean

Напоминаем, что в дебаггере вы всегда можете выделить мышкой выражение и посмотреть на его значение:

```
1 function calculateCommission(input) { input = 10
2   const percent = 0.01; percent = 0.01
3   const minCommission = 0.35; minCommission = 0.35
4   let commission = input * percent; commission = 0.1, input = 10,
5   if (commission < minCommission) { minCommission = 0.35
6   commission = minCommission;
7 }
8 return commission;
9 }
```



boolean advanced

Мы с вами говорили, что в JS есть приведение типов. Так вот на самом деле, не обязательно, чтобы внутри скобок было выражение именно типа `boolean`. Там может быть любое выражение, но существует одно простое правило: если значение этого выражения вычисляется в одно из следующих, то оно "воспринимается" как `false`:

1. `0`
2. `""`, `" "`, ``` – пустая строка (внутри нет ничего, даже пробела)
3. `null`
4. `undefined`
5. `NaN`

Эти значения (включая само `false`) называют falsy-values. Их список фиксирован (хотя в новой версии добавилось ещё одно – `0n`) и для собеседования его (список) желательно выучить.



Несколько return

Для того, чтобы поменять комиссию, мы с вами использовали `let`, т.к. `const` не позволяет привязывать другие значения. При этом сами же мы говорили, что лучше стараться везде использовать `const`. Можно ли найти выход из этой ситуации?



Несколько return

Оказывается, можно внутри функции использовать не один `return`, а несколько:

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   }  
8   return commission;  
9 }
```

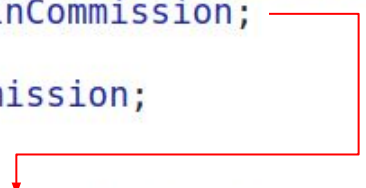
Пройдитесь дебаггером по коду, и посмотрите, что будет, если вы попадёте на 6-ую строку.



Как это работает?

Когда браузер встречает `return` внутри функции – это для него сигнал того, что "дело сделано", можно выходить из функции. Т.е. попав на 6-ю строку, мы уже не будем выполнять 8-ую строку, а сразу перейдём на 11-ую (в присваивание):

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   }  
8   return commission;  
9 }  
10  
11 const result = calculateCommission(10);  
12 console.log(result);
```



Как это работает?

Как себе это представлять: например, вы приходите в большой магазин за определённой вещью (кофе), и вам улыбается удача, эта вещь находится не где-то в глубине магазина, а прямо рядом со входом и кассами.

Поскольку вы торопитесь, вы сразу берёте эту вещь и идёте на кассу (так поступают, конечно, не все, но это уже детали).



if, else и т.д. advanced

В книжках вы можете встретить описание конструкций `else` и т.д. Мы пока их не рассматриваем по одной простой причине – они нам не нужны. То, как мы построили нашу функцию с успехом заменяет использование многих конструкций.

Но для полноты картины давайте сравним:

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   }  
8   return commission;  
9 }
```

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   } else {  
8     return commission;  
9   }  
10 }
```



if, else и т.д. advanced

`else` используется с `if` и означает следующее: если выражение в скобках к `if` вычислилось в `falsy`, тогда выполняй не блок `if`, а выполняй блок `else`.

Но: строк стало больше и читать стало сложнее – это плохо. Чем проще читать ваш код, тем лучше. Профессионализм программиста заключается в умении писать простой код для решения сложных задач (а не наоборот).

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   }  
8   return commission;  
9 }
```

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission) {  
6     return minCommission;  
7   } else {  
8     return commission;  
9   }  
10 }
```



Блоки в if advanced

Кроме того, вы можете встретить информацию о том, что для `if` можно не создавать блок кода и писать просто вот так:

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4   const commission = input * percent;  
5   if (commission < minCommission)  
6     return minCommission;  
7   return commission;  
8 }
```

Это плохой подход (на нашей практике он всегда приводил к тому, что практикующий его программист потом тратил часы в дебаггере, пытаясь понять, почему "всё глючит"). Поэтому робот, проверяющий ДЗ, такой код принимать не будет (вам придётся переделывать).



Блоки в if advanced

Q: почему этот код плох?

A: потому что к `if` будет относиться ровно одна строка, а вы можете подставить туда ещё одну:

```
function calculateComission(input) {  
  const percent = 0.01;  
  const minCommission = 0.35;  
  const commission = input * percent;  
  if (commission < minCommission)  
    console.log(minCommission);  
    return minCommission;  
  return commission;  
}
```

И пусть даже визуально кажется, что к `if` относятся две строки, в реальности мы получим вот такой код:

```
if (commission < minCommission)  
  console.log(minCommission);  
return minCommission;  
return commission;
```

Т.е. независимо от суммы, мы всегда будем возвращать минимальную комиссию (и потеряем много денег)



Блоки в if advanced

Поэтому настоятельно рекомендуем для любой конструкции (`if`, `else` и т.д.) писать блок, поскольку на нашем опыте любой, кто с "хвастовством" утверждал, что уж он-то никогда не допустит ошибку (как в примере на предыдущем слайде), потом жаловался, что не поймёт, где ошибка.



Пустые строки

И последнее - не жалейте пустых строк для улучшения читабельности кода:

```
1 function calculateCommission(input) {  
2   const percent = 0.01;  
3   const minCommission = 0.35;  
4  
5   const commission = input * percent;  
6   if (commission < minCommission) {  
7     return minCommission;  
8   }  
9  
10  return commission;  
11 }
```

Вот так код выглядит гораздо лучше (пусть и стало на целых две строки больше).



ИТОГИ



ИТОГИ

В этой лекции мы обсудили достаточно много важных моментов:

1. Создание функций
2. Условия
3. `undefined` и `boolean`

Мы намеренно не рассматриваем некоторые конструкции (например, `else`, тернарный оператор, `switch`) до тех пор, пока они нам не пригодятся (потому что пока нет смысла их учить – без реального применения вы их быстро забудете).

А некоторые не будем рассматривать вовсе, поскольку их можно заменить тем, что мы уже знаем, ведь наша цель – не выучить досконально всё, а как можно раньше получить результат (научиться что-то делать).



ДОМАШНЕЕ ЗАДАНИЕ



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.


Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

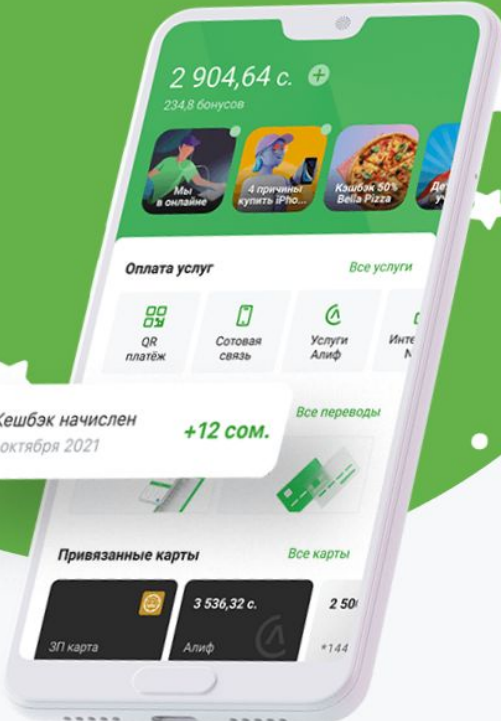
Все вопросы вы сможете задавать в [Телеграм канале](#).



ДЗ №1: Кэшбэк

Давайте посмотрим на сервис <https://cashback.alif.tj>:





The smartphone screen shows the Alif Mobi app interface. At the top, it displays a balance of 2,904.64 CMT and 234.8 bonuses. Below this, there are three promotional banners: 'Мы в онлайн', '4 причины купить iPhone', and 'Кэшбек 50% Bella Pizza'. The main menu includes 'Оплата услуг' (Payment of services) with options for QR payment, mobile communication, Alif services, and internet, and 'Все услуги' (All services). A notification bubble states 'Кэшбэк начислен 7 октября 2021 +12 сом.' (Cashback credited 7 October 2021 +12 CMT). Below the notification, there is a section for 'Привязанные карты' (Linked cards) showing two cards: 'ЗП карта' (3,536.32 CMT) and 'Алиф' (2,500 CMT). The bottom right corner shows a card number ending in *144.

**Выгодные покупки
кэшбэк до 30%**



ДЗ №1: Кэшбэк

Правила сервиса гарантируют вам с любой покупки получение кэшбэка в зависимости от магазина.

Переделываете своё предыдущее ДЗ

Вам необходимо написать **функцию**, которая исходя из суммы покупки и кэшбэка конкретного магазина высчитывает итоговый кэшбэк. Важно: округлять до сомони не нужно (мы разберём вопросы округления позже).



ДЗ №1: Кэшбэк

Чтобы автоматизированная система смогла проверить вашу задачу, необходимо выполнить ряд требований:

1. Внутри архива должен быть каталог `cashback`
2. Название функции должно быть `calculateCashback` и первым параметром содержать сумму, а вторым – процент магазина:

```
function calculateCashback(amount, percent) {  
  // ваш код  
  
  return cashback;  
}
```

Нужно вынести в переменные

```
const result = calculateCashback(1000, 50);  
console.log(result);
```

Важно: функция обязательно должна возвращать результат.



ДЗ №2: РКО

Нас интересует следующая услуга:

Переводы

Исходящие переводы на счета в других банках в иностранной валюте:

в долларах США

в рамках сетки основного банка корреспондента

0,2% от суммы

(мин. 250, макс. 450 сомони)

Напишите функцию, которая по входной сумме рассчитывает итоговую комиссию.



ДЗ №2: РКО

Чтобы автоматизированная система смогла проверить вашу задачу, необходимо выполнить ряд требований:

1. Внутри отправляемого архива должен быть каталог **rko**
2. Название функции должно быть **calculateCommission** и содержать два параметра – сумму перевода в долларах и курс доллара к сомони*:


```
function calculateCommission(amount, rate) {  
  // ваш код  
  
  return comission;  
}  
  
const result = calculateCommission(1000, 11.40);  
console.log(result);
```

Функция обязательно должна возвращать результат



ДЗ №3: Снятие наличных

Представим, что в alif.mobi есть услуга снятия наличных в кассе банка. Условия достаточно простые и выглядят следующим образом:

Операции		
	Получение наличных в кассе Банка (до 5 000 сомони в календарный месяц)	0%
	Получение наличных в кассе Банка (свыше 5 000 сомони в календарный месяц)	0,7% (мин. 0,35 с.)

Напишите функцию, которая по текущей запрашиваемой сумме и сумме уже снятых в этом месяце денег рассчитывает итоговую комиссию.



ДЗ №3: Снятие наличных

Чтобы автоматизированная система смогла проверить вашу задачу, необходимо выполнить ряд требований:

1. В архиве должен быть каталог `mobi`
2. Название функции должно быть `calculateCommission` и содержать два параметра – сумму текущего снятия (сколько клиент хочет снять сейчас) и сумму предыдущих снятий (сколько он уже снял в этом месяце, без учёта текущей суммы):

```
function calculateCommission(amount, total) {  
  // ваш код  
  
  return comission;  
}
```

```
const result = calculateCommission(1000, 0);  
console.log(result);
```

Функция обязательно должна возвращать результат



ДЗ №3: Снятие наличных

Важно: будем считать, что при сумме свыше 5000 сомони, комиссия 0.7% считается с суммы, превышающей 5000 сомони.



Спасибо за внимание

alif skills

2023г.

